Simon Schliecker,
Arne Hamann,
Razvan Racu,
Rolf Ernst

Formal Methods for System Level Performance Analysis and Optimization

Braunschweig : Institute of Computer and Communication Network Engineering, 2009

Formal Methods for System Level Performance Analysis and Optimization

Simon Schliecker, Arne Hamann, Razvan Racu, Rolf Ernst
Institute of Computer and Communication Network Engineering
Technical University of Braunschweig
{schliecker|hamann|racu|ernst}@ida.ing.tu-bs.de

***Abstract*— With increasing system complexity, there is growing interest in using formal methods in wider range of systems to improve system predictability and determine system robustness to changes, enhancements and pitfalls. This paper gives an overview over a formal approach to system level performance modelling and analysis. A methodology is presented to cover distributed multiprocessor systems as well as multiprocessor systems on chip. The abstract modelling allows early design space exploration and optimization. We investigate an example multimedia application and optimize the usage of the shared memory to reach an optimal performance.**

I. Introduction

Formal approaches to system performance modelling have always been used in real-time systems design. With increasing system complexity, there is growing interest in using formal methods in wider range of systems to improve system predictability and determine system robustness to changes, enhancements and pitfalls.

Significant progress has been made in performance modelling and analysis in the last couple of years. Worst case timing analysis of individual software processes is now in a state where it can be applied to advanced architectures with caches with the large conservative overestimation of earlier years. Early industrial adopters are found in the aircraft industry where WCET analysis has been included in the regular design process.

These advances are accompanied by new modular models and methods that allow to analyze large scale, heterogeneous systems, providing reliable data on transitional load situations, end-to-end timing, memory usage, or packet losses. The corresponding methods and tools are now regularly used in automotive design at early industrial adopters. There, analysis is combined with tracing and simulation to cover the difficult corners of the system state space resulting from parallel execution in distributed applications and communication over heterogeneous networks. It is also used for early evaluation of architectures with respect to extensibility or flexibility in combination with design space exploration support.

In addition to large scale distributed systems, formal performance analysis methods are also becoming increasingly important in the domain of tightly integrated Multiprocessor-Systems-On-Chips (MpSoC). Such components promise to deliver higher performance at a reduced production cost and power consumption, but they also introduce a new level of integration complexity. Like in distributed embedded systems, multiprocessing comes at the cost of higher timing complexity of interdependent computation and communication.

This paper presents a tutorial overview over such a modular formal performance analysis framework. We present the basic procedure (Sec. II) and the necessary extensions to specifically address the timing interdependencies of multi-core architectures, such as accesses to a shared memory (Sec. III). We present a methodology to systematically explore the design space for optimal configurations (Secs. IV and V). In an experimental section (Sec. VI), we investigate an example heterogenous SoC architecture for timing bottlenecks, and show how to improve the performance guided by sensitivity analysis and system exploration.

II. Formal Multiprocessor Performance Analysis

In the past years, *compositional performance analysis* approaches [9], [4], [10] have received increasing attention in the real-time systems community. Compositional performance analyses exhibit great flexibility and scalability for timing and performance analysis of complex distributed embedded real-time systems. Their basic idea is to integrate local performance analysis techniques, e.g. scheduling analysis techniques known from real-time research, into system level analyses. This composition is achieved by connecting the component's inputs and outputs by stream representations of their communication behavior using event models.

1) Application model: An embedded system consists of hardware and software components interacting with each other to realize a set of functionalities. The traditional approach to formal performance analysis is performed bottom-up. First, the behavior of the individual functions is investigated in detail to gather all relevant data such as the execution time. This information can then be used to derive the behavior within individual components, accounting for local scheduling interference. Finally, the system level timing is derived on the basis of the lower level results.

For efficient system level performance verification, embedded systems are modeled with the highest possible level of abstraction. The smallest unit modeling performance characteristics at the application level is called *task*. Furthermore, to distinguish computation and communication, tasks are categorized into computational and communication tasks. The hardware platform is modeled by *computational* and *communication resources*, that are referred to as CPUs and

buses, respectively. Tasks are mapped on resources in order to execute. To resolve request conflicts each resource is associated with a *scheduler*.

Tasks are activated and executed due to activating events, that can be generated in a multitude of ways, including timer expiration, and task chaining according to inter-task dependencies. Each task is assumed to have one input FIFO. In the basic task model (see Figure 3a), a task reads its activating data solely from its input FIFO and writes data into the input FIFOs of dependent tasks. Besides this basic activation model, some approaches have been extended to allow for more complex activation semantics [5].

2) Event streams: Timing properties of the activation model describe the arrival of workload, i.e. activating events, at the task inputs. Instead of considering each activation individually, as simulation does, formal performance verification abstracts from individual activating events to *event streams*. Note that different methods utilize different stream models to describe the timing of activating events. Generally, event streams can be described using the upper and lower event arrival functions η^+ and η^-.

Definition 1 (Upper Event Arrival Function η^+): The upper event arrival function $\eta^+(\Delta t)$ specifies the maximum number of events that occur in the event stream during any time interval of length Δt.

Definition 2 (Lower Event Arrival Function η^-): The lower event arrival function $\eta^-(\Delta t)$ specifies the minimum number of events that occur in the event stream during any time interval of length Δt.

Correspondingly, an event model can also be specified using the functions $\delta^-(n)$ and $\delta^+(n)$ that represent the minimum and maximum distance between any n events in the stream.

For computational efficiency, event models can be represented with various parameters. One example are the standard event models capturing key properties of event streams using three parameters: period, jitter, and minimum distance. Figure 1 visualizes the upper and lower event arrival functions (η^+ and η^-) for standard event models.

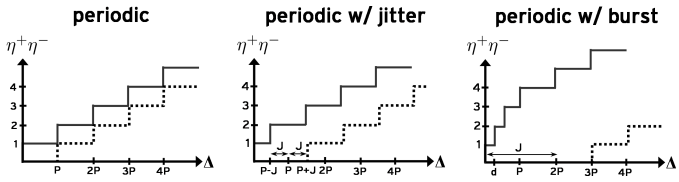

Fig. 1. Standard event models

3) Local component analysis: Based on the underlying resource sharing strategy as well as stream representations of the incoming workload modeled through so-called *activating event models*, local component analyses systematically derive worst-case scenarios to calculate worst-case (sometimes also best-case) task response times (BCRT, WCRT), i.e. the time between task activation and task completion, for all tasks sharing the same resource. Thereby, local component analyses guarantee that all observable response times fall into the calculated [best-case, worst-case] interval. These analyses are therefore called *conservative*.

Note that different approaches use different models of computation to perform local component analyses. SymTA/S, for instance, is based on the algebraic solution of so-called response time formulas using the *sliding window technique* proposed by Lehoczky [11], whereas the Real-Time Calculus utilizes arrival curves and service curves to characterize workload and processing capabilities of components, and determine their real-time behavior [4]. These concepts are based on the so-called *network calculus*. For details please refer to [3].

Additionally, local component analyses determine the communication behavior at the outputs of the analyzed tasks by considering the effects of scheduling. Therefore, it is the basic model assumes that tasks produce output events at the end of each execution. Like the input timing behavior, also the output timing behavior is captured by event models. Mainly based on the local response time analysis, the *output event models* can then be derived for every task. For events that are generated during the execution of a task (such as coprocessor calls or memory accesses) consider Section III.

4) Compositional system level analysis loop: The basic idea of the compositional system level analysis is visualized on the right hand side of Figure 2, see e.g. [9], [4] (the shared resource analysis will be explained in Sec. III).

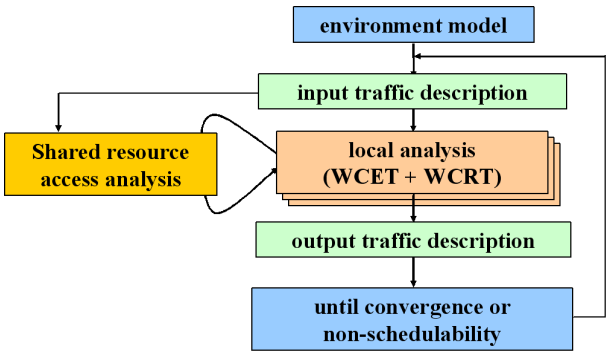

Fig. 2. MpSoC Performance Analysis Loop

Compositional system level analysis alternates local component analysis (as explained in Section II-.3) and output event model propagation. More precisely, in each global iteration of the compositional system level analysis, local analysis is performed for all component to derive response times and output event models. Afterwards, the calculated output event models are propagated to the connected components, where they are used as activating event models for the subsequent global iteration. Obviously, this iterative analysis represents a fix-point problem. If after an iteration all calculated output event models stay unmodified, convergence is reached and the last calculated task response times are valid.

To successfully apply compositional system level analysis, the input event models of all components need to be known or must be computable by local component analysis. Obviously, for systems containing feed-back between two or

more components this is not the case, and thus system level analysis cannot be performed without additional measures. The concrete strategy to overcome this problem depends on the component types and their input event models. One possibility is the so-called *starting point generation* of SymTA/S [9].

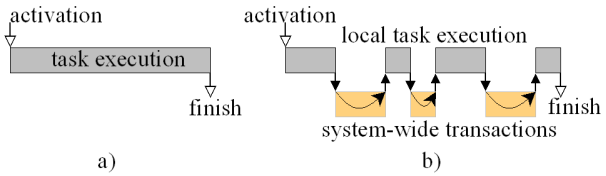

Fig. 3. Task Execution Model.

III. From Distributed Systems to MPSoCs

This procedure has been extended in [14] to account for shared memory systems. The model of the task behavior is extended to include local execution and memory transactions during the execution. Such a *communicating task* performs *transactions* during its execution as depicted in Figure 3b. The depicted task requires three chunks of data from an external resource. It issues a *request* and may only continue execution after the transaction was e.g. transmitted over the bus, processed on the remote component and transmitted back to the requesting source. Such memory accesses may be explicit data fetch operations or implicit cache misses.

Such memory accesses, especially cache misses, are extremely difficult to predict precisely. Therefore the analysis can not with passable effort predict the timing of each transaction. Instead a shared resource access analysis algorithm is introduced that subsumes all transactions of a task execution and the interference by other system activities. We will outline and exploit that model in Section III-C.

The memory is considered as a separate component and an analysis must be available for it to predict the timing of a set of memory requests. For this analysis to work, the event models for the amount of requests issued from the various processors are required. The outer analysis procedure will provide these event models throughout the system. The processor scheduling analysis can account for memory access timing by calling the memory analysis with locally derived memory event models and additional information (addresses and time frames). This is shown on the left hand side of Figure 2.

In order to embed the analysis of communicating tasks into the compositional analysis framework described in Sec. II, three major building blocks are required

1. Deriving the amount of transactions issued by a task and all tasks on a processor.
2. Deriving the latency experienced by a set of transactions.
3. Integrating the transaction latency into the worst-case response time.

These three steps will be presented in the following. We begin with the local investigation of derving the amount of initiated transactions (Sec. III-A) and the extended worst-case response time analysis (Sec. III-B). Finally, we turn to the system level problem of deriving the transaction latency (Sec. III-C).

A. Deriving Output Event Models

The shared memory access delay that an invocation of task τ_i can experience during its execution depends on the amount of requests it issues and (indirectly via the conflicts on the shared memory) on the amount of transactions issued by other tasks accessing the same shared memory.

For each task the amount of issued transactions may be bounded by closely investigating the tasks control flow that contains the logical structure of the task execution (consisting of a set of “basic blocks” of linear code that are connected with edges representing all possible execution sequences). For example, a task may explicitly fetch data each time it executes a for-loop that is repeated several times. By multiplying the maximum number of loop iterations with the amount of fetched data a bound on the memory accesses can be derived. Implicit data fetches (cache misses) can also be handled, by using approaches such as [16] that identify for each basic block the maximum number of cache misses that may occur during the execution. The approach in [15] performs a detailed analysis, not only bounding the total number of memory accesses per task execution, but also deriving conservative bounds on their minimum distances.

This procedure allows to conservatively derive the event model functions $\eta_\tau^+(w)$ and $\eta_\tau^-(w)$ that represent the request traffic that each task τ in the system can produce within a given time window of size w. Communicating tasks that share the same processor may be executed alternately, resulting in a combined request traffic for the complete processor. This again can be expressed as an event model. E.g. a very simple approach is to approximate the processor’s request traffic (in a given time window) with the sum of the request traffic of each task executing on that processor. Obviously, this is an overestimation, as the tasks will not execute at the same time. An amelioration can be found in [15], where the exclusive execution is considered.

For the purpose of this paper, we differentiate between different types of transactions. For example one type of transaction may be a cache miss that results in the fetching of one complete memory row accessed as a burst of 8 words. Another type of transactions may be the fetching of a macroblock or a complete frame. Although all types of transactions could in this case be expressed as a set of elementary memory accesses, this procedure allows convenient modeling and hiding of implementation details (as demonstrated in the experiments).

B. Response Time Analysis with Memory Accesses

Memory access delays may be treated differently by various processor implementations. Many processors, and some among the most commonly used, allow tasks to perform coprocessor or memory accesses by offering a multi-cycle operation that stalls the complete processor until the transaction has been processed by the system. In other cases, a set of hardware threads may allow to perform a quick context switch to another thread that is ready, effectively keeping the processor utilized. While this behavior usually has a beneficial effect on the average throughput of a system, multithreading is discouraged

in priority based systems with reactive or control applications. There the worst-case response time of high priority tasks may actually decrease. This has been investigated in [14].

The integration of dynamic memory access delays into the real-time analysis will in the following be performed for a processor with priority based preemptive scheduling that is stalled during memory accesses. In such a system a task's worst case response time is determined by the task's worst case execution time plus the maximum amount of time the task can be kept from executing due to preemptions by higher priority tasks and blocking by lower priority tasks. A task that performs memory accesses is additionally delayed when waiting for the arrival of requested data. Furthermore, memory accesses by high priority tasks will cause lower priority tasks to be preempted for a longer amount of time.

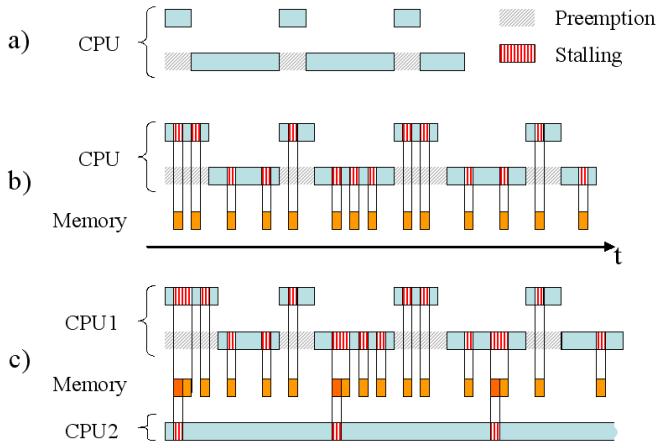

Fig. 4. Tasks on different Processors accessing a Shared Memory. a) and b) Single Processor Case, c) Conflicts from another CPU.

This is depicted in Figure 4. In the case when both tasks execute in the local memory (Scenario 4a) the low priority task is kept from executing by three invocations of the high priority tasks. When both tasks access the same memory (Scenario 4b), each of the 3 preemptions will take a longer amount of time (due to the processor being stalled when the higher priority task accesses the memory). Also, the low priority task itself fetches data from the memory, causing a larger execution requirement. These two effects will cause the response time of the low priority task to grow so much, that it suffers from an additional preemption of the other task, which possibly jeopardizes a given deadline.

On the basis of these observations, a response time equation can be derived for the example scheduler. The response time is the sum of the following:

- The core execution times of all tasks mapped to the processor and their activation event models.
- The increased context switch time due to the resources being stalled during memory accesses.
- The delay caused by the memory accesses, which is a function of the memory accesses of a specific task and the higher priority tasks. This is investigated in Section III-C.

Variations of such a response time analysis have been presented for single- and multithreaded static priority preemptive scheduling [14], as well as for round-robin scheduling. Other scheduling policies for which classical real-time analysis is available can be straight-forwardly extended by including a term that represents the accumulated busy time.

C. Deriving Accumulated Busy Times

Deriving the timing of many memory accesses has recently become an important subject in real-time research. Previously, the worst case timing of individual events was the main concern. Technically, a sufficient solution to find the delay that many events experience, is to derive the single worst-case load scenario and assume it for every access. However, not every memory request will experience a worst case system state, such as worst case time wheel positions in TDMA, or transient overloads in priority based components. For example, the task on CPU2 in Figure 4 will periodically access the shared memory, and as a consequence disturb the accesses by the two tasks on CPU1. A "worst case memory access" will experience this delay, but of all accesses from CPU1, this only happens at most 3 times in the example. Thus, accounting this interference for every memory access leads to very unsatisfactory results — which has previously prevented the use of conservative methods in this context.

The key idea is instead to consider all requests during the lifetime of a task jointly. We introduce the worst case *accumulated busy time*, which is defined as the total amount of time during which *at least one* request has been issued but is not finished. Many requests in a certain amount of time can *in total* only be delayed by a certain amount of interference, which can be straightforwardly covered by the accumulated busy time analysis.

This accumulated busy time can be efficiently calculated e.g. for a shared bus: A set of requests is issued from different processors that may interfere with each other. The exact individual request times are unknown and their actual latency is highly dynamic. Extracting detailed timing information (e.g. when a specific cache miss occurs) is virtually impossible, and considering such details in a conservative analysis is highly exponential. Consequently, we waive such details and focus on bounding the accumulated busy time. Given a certain dynamism in the system, this consideration will not result in excessive overestimations.

Without bus access prioritization, it has to be assumed that it is possible for every transaction issued by any processor during the lifetime of a task activation i that they will disturb the transactions issued by i. In the present setup this is given by the requests issued by the other concurrently active tasks on the other processors, as well as the tasks on the same processor as their requests are treated first-come-first-served.

Given a set of requests Q_i that are sent from the same processor and treated first-come-first-served on the memory, their accumulated busy time can be bounded as follows:

$$S(Q_i, w) \quad \leq \quad |Q_i| \cdot C_i + \sum_{p \in P} \sum_{\tau \in p} \eta_\tau^+(w) \cdot C_\tau \qquad (1)$$

$|Q_i|$ is the number of requests sent from the same processor within a time window of size w. C_i is the amount of time the memory requires to process one of these requests.

P is the set of *other* processors in the system and τ is a task mapped to a processor p.

$\eta_\tau^+(w)$ is the maximum number of requests sent by task τ within a time window of size w, and C_τ is the amount of time required to process one of these requests.

If a memory controller is utilized, this can simply be considered in the above equation. For example, all requests from a certain processor may be prioritized over those of another. Then, C_τ is 0, if the corresponding requests receive a lower priority. Additionally, a small blocking factor of one elementary memory access time is required, to model the time before a transaction may be aborted for the benefit of a higher priority request.

The compositional analysis approach of Section II used together with the methods of Section III now delivers a complete framework for performance analysis of heterogeneous multiprocessor systems with shared memories.

IV. System Exploration and Optimization

In this section techniques for automated design space exploration and system optimization are discussed. First, a design space exploration framework that is based on stochastic optimization techniques is shortly introduced (Section IV-A, for details please refer to [6]). This framework can be applied, for instance, to optimize priority and time slot assignments in distributed embedded systems. Another key property that is exploited by the proposed design space exploration framework is the adaptation of event timing in streams connecting functionally dependent components. This technique, called *traffic shaping*, is discussed in Section IV-B. In the experimental Section VI, traffic shaping will be systematically used to optimize the given example architecture.

A. Design Space Exploration Framework

Figure 5 visualizes the functionality of the utilized design space exploration framework [6].

The *Optimization Controller* is the central element. It is connected to the *Analysis Engine* and to the *Evolutionary Optimizer*. The *Analysis Engine* checks the validity of given parameter configurations and provides data for the *Objectives* to calculate the fitness values subject to optimization.

The *Evolutionary Optimizer* is responsible for the problem-independent part of the optimization problem, i.e. elimination of poor parameter configurations and selection of interesting parameter configurations for variation. Currently, SPEA2 (Strength Pareto Evolutionary Algorithm 2) [17] is used for this part, that is coupled via the PISA interface (Platform and Programming Language Independent Interface for Search Algorithms) [2] to the exploration framework. Note that the selection and elimination strategy, i.e. the strategy to walk through the search space and to approximate the Pareto-optimal solution set in case of multi-criterion optimization, depends on the utilized optimizer.

The system is seen component wise for design space exploration. Modifiable parameters of different physical or logical components are encoded separately by specialized *chromosomes*. Chromosomes contain problem-aware variation operators guiding the search process for the system part they represent (i.e. crossover and mutation). Consequently, chromosomes are responsible for the problem-specific part of the optimization problem. Specific chromosomes tailored for timing and performance exploration of distributed embedded systems are discussed in [6].

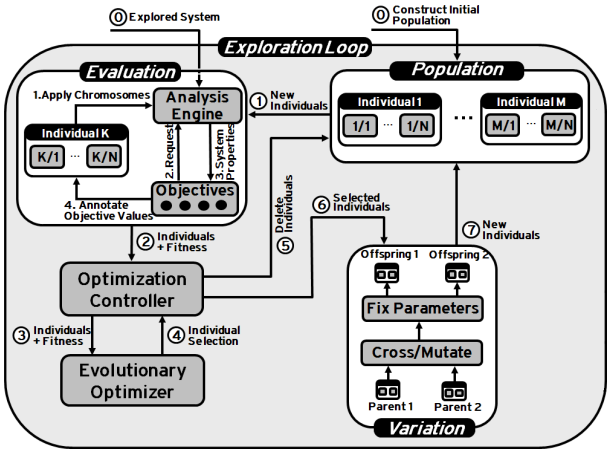

Fig. 5. Design space exploration loop

Before exploration can be started *Chromosomes* representing the desired search space as well as *Objectives* subject to optimization have to be selected and configured. Selected chromosomes are included into evolutionary exploration, while all other parameters remain immutable. After specification of the optimization task, the *Analysis Engine* is initialized with the immutable part of the search space, and the selected chromosomes are used as blueprints to create the initial population (step 0). In other words, each parameter configuration consists of specific chromosome instances (phenotypes).

Afterwards, each non-evaluated parameter configuration is evaluated (step 1). First, the parameter configuration's chromosome instances are applied to the *Analysis Engine* (Step 1.1). This completes the system and it is analyzed. Afterwards, each *Objective* requests necessary system properties to calculate its fitness value (Steps 1.2 and 1.3). Finally, the parameter configuration is annotated with the calculated fitness values (Step 1.4).

Once all parameter configurations have been analyzed, they are communicated along with their fitness values to the *Optimization Controller* (step 2) that forwards this information to the *Evolutionary Optimizer* (step 3). Based on the fitness values the *Evolutionary Optimizer* creates two lists, a list of parameter configurations selected for deletion and a list of parameter configurations selected for variation, and sends

them back to the *Optimization Controller* (step 4). Afterwards the *Optimization Controller* manipulates the population. First, parameter configurations selected for deletion are removed from the population (step 5). Second, parameter configurations selected for variation are used to create new parameter configurations through recombination (i.e. mutation and crossover, step 6). Finally, all created parameter configurations are added to the population (step 7). This completes the processing of one generation, and the whole loop begins again.

B. Optimization through traffic shaping

Scheduling and data dependent behavior induce jitter to the input-output timing of processes and communication [9]. Such jitters accumulate in the system and can lead to event bursts. Both effects increase timing uncertainty and worst-case peak load. Peak loads caused by bursty streams can be controlled by modulating the maximum number of events per time. This technique, called *traffic shaping*, can reduce the global impact of peak loads at the cost of increased latencies along the manipulated event streams. The shaping effects are rather complex and require special modeling considerations that are briefly explained in the following.

Traffic shaping consists in enforcing bounds on minimum event distances in streams connecting functionally dependent components using time-out buffers. More precisely, the time-out mechanism buffers incoming events such that two successive events are not released earlier in time than $d^-_{\text{time out}}$.

According to the extended real-time calculus approach of Thiele et al. [4], the shaper defines a sporadic upper-bound *service curve*:

$$\eta^+_{\text{time out}}(\Delta t) = \left\lceil \frac{\Delta t}{d^-_{\text{time out}}} \right\rceil$$

The shapers output arrival curve can be calculated from both, input arrival curve $\eta^+_{\text{in}}(\Delta t)$ and shaper service curve $\eta^+_{\text{time out}}(\Delta t)$. In case of traffic shapers the real-time calculus equations can be simplified to

$$\begin{aligned} \eta^+_{\text{shaped}}(\Delta t) &= \min\left(\eta^+_{\text{timeout}}(\Delta t), \eta^+_{\text{in}}(\Delta t)\right) \\ &= \min\left(\left\lceil \frac{\Delta t}{d^-_{\text{timeout}}} \right\rceil, \left\lceil \frac{\Delta t}{d^-_{\text{in}}} \right\rceil, \left\lceil \frac{\Delta t + J}{P} \right\rceil\right). \end{aligned}$$

Figure 6 illustrates the effect of traffic shaping. The arrival curve with minimum distance d^-_{in} covers the service curve defined by $d^-_{\text{time out}}$. The block arrows indicate buffering.

The vertical distance between arrival and service curve captures the so-called backlog [4], i.e. the number of buffered events at a given point in time. Correspondingly, the horizontal distance between the curves, i.e. the arrow lengths in Figure 6, represents the buffering delay. In order to obtain conservative values, the maximum horizontal and vertical distances need to be determined. Details can be found in [9].

V. Sensitivity Analysis

Complementary to system exploration and optimization, sensitivity analysis provides information about the robustness

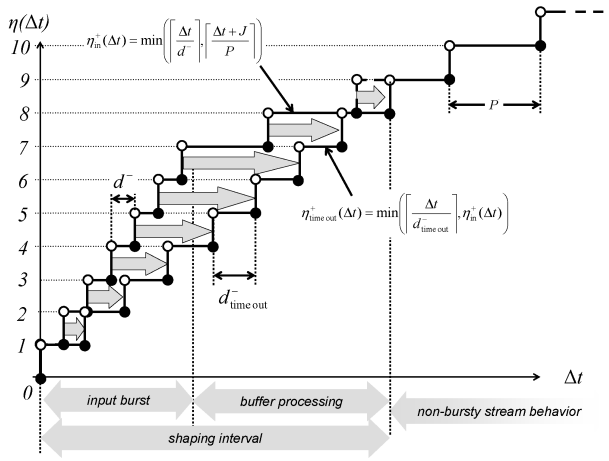

Fig. 6. Event arrival curve of output event stream

of the system properties with respect to performance constraints. The system properties represent system characteristics indirectly determined by the specification of the HW and SW components. These include, the execution/communication time intervals of the computational/communication tasks, the timing parameters of the tasks' activation models, or the speed factor of HW resources. Given an initial system configuration, sensitivity analysis determines the maximum variation of the system properties that the system can accomodate. This variation is referred to as *performance slack*. Based on the design strategy, two scenarios to utilize the performance slack are identified:

a) System dimensioning: To reduce the global system cost, important especially for systems with short life-time or portable systems, the system designer can decide to use the performance slack for efficient system dimensioning. In this case, instead of looking for system configurations that can accomodate later changes, the performance slack is used to optimize the system cost by selecting cheaper variants for processores, communication resources or memories. More complex scenarios even imply the integration of the entire application on alternative platforms, reducing the number of hardware components [12]. Note that, lower cost implies on one side lower hardware costs, and on the other side lower operability costs, like power or size.

b) System robustness optimization: Based on the slack values, the designer defines a set of robustness metrics to cover different possible design scenarios. In order to maximize the system robustness, the defined robustness metrics are used as optimization objectives by automatic design space exploration and optimization tools [7]. The scope is to obtain system configurations with less sensitivity to later design changes.

Our sensitivity analysis approach is based on a binary search technique, ensuring full compatibility with different performance analysis engines, and the transparent implementation with respect to application structure, system architecture and scheduling algorithms. A detailed description of the sensitivity analysis algorithms for different system properties can be found in [13].

VI. Experiments

In this section, the formal methods presented in this paper are applied to the example system depicted in Figure 7. The system consists of 3 processing units. A digital signal processor and a configurable hardware component execute a media processing application. The DSP decodes frames of an incoming media stream that are successively post-processed on the dedicated hardware, before they are displayed to the end user. Additionally, a RISC CPU filters IP traffic that is embedded into the media stream (datacast). During operation, the hardware can be dynamically reconfigured by the CPU to perform denoise, rescaling, or other post processing operations. During the reconfiguration process, that Hardware cannot be used.

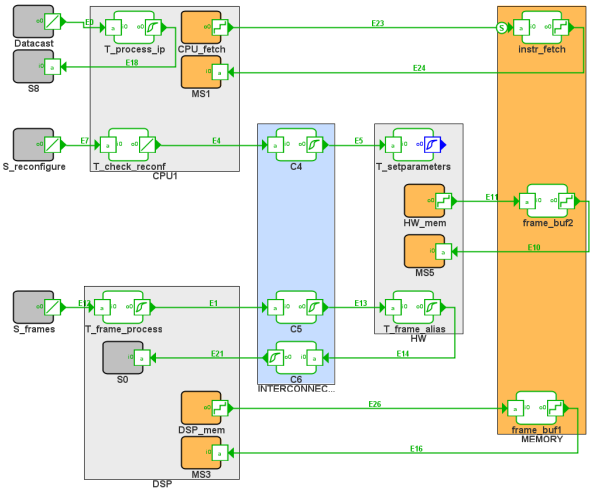

Fig. 7. Application Model

Core execution and communication times as well as memory accesses (number of accesses and time per access sequence) of all tasks in the system are specified in Table I. The system is subject to two end-to-end latency constraints. The latency of the datacast application ($Datacast \rightarrow S8$) may not exceed 730 ms, whereas frame processing ($S\_frames \rightarrow S0$) must be finished within 550 ms to meet the throughput requirement of the media application.

A. Local Memories

Initially assume that the CPU, the Hardware, and the DSP have been used in previous product generations. In the initial development phase they are now integrated to perform the new applications. To allow a seemless integration, all hardware components dispose of sufficiently large local memories to perform their operations and to buffer data. In this initial setup, our approach deems the system well schedulable (340 ms for $Datacast \rightarrow S8$ and 272 ms for $S\_frames \rightarrow S0$).

B. Globally Shared Memory

We will now explore the option of sharing the memory between the different components, which can be beneficial to save cost and increase the data sharing efficiency between the Hardware and the DSP. Obviously, this causes conflicting shared memory accesses between the DSP and the Hardware to store, load, and exchange intermediate results. Additionally, the CPU must regularly fetch intructions from shared memory into the local cache. Given first-come-first-served processing of the various memory accesses, the uncontrolled competition for the memory causes all latency constraints to be violated (1135 ms for $Datacast \rightarrow S8$ and 752 ms for $S\_frames \rightarrow S0$).

Task Name	Core Execution Time	Memory Accesses	Communication
T_process_ip	100 ms	14*10 ms	-
T_check_reconf	100 ms	-	100 ms
T_set_parameters	2 ms	16*5 ms	-
T_frame_process	20 ms	10*10 ms	10 ms
T_frame_post_process	10 ms	6*5 ms	10 ms

Table I. Core execution / communication times and memory accesses (maximal burst size + time for one access sequence)

To counter this effect, smart memory controllers are used to balance the traffic requirements and bandwidth between the requesting sources. In our setup, we assume a memory controller that can prioritize traffic from different sources. Furthermore, the memory controller can limit the amount of traffic processesed per source by introducing a minimum distance between requests as explained in Sec. IV-B. Similar memory controllers have been proposed in [8] and [1].

C. Optimizing the design

First, it is tried to find feasible system configurations by optimizing the priorities of the shared memory accesses to $intr\_fetch$, $frame\_buf\_1$, and $frame\_buf\_2$. By applying design space exploration, four different Pareto-optimal (with respect to end-to-end latencies) priority assignments are found (Table II).

Priorities	$Datacast \rightarrow S8$	$S\_reconfigure \rightarrow T\_setparameters$	$S\_frames \rightarrow S0$
$inst\_fetch > frame\_buf1 > frame\_buf2$	340 √	612 √	752×
$frame\_buf1 > frame\_buf2 > inst\_fetch$	940 ×	302 √	537 √
$frame\_buf2 > frame\_buf1 > inst\_fetch$	940 ×	432 √	412 √
$frame\_buf2 > inst\_fetch > frame\_buf1$	1135×	302 √	392 √

Table II. Pareto-optimal (infeasible) parameter configuration obtained through optimization of shared memory access priorities

As can be observed, none of the obtained parameter configurations results in a feasible system. The first configuration violates the latency constraint for the frame processing application ($S\_frames \rightarrow S0$), whereas the remaining three configurations result in too high processing times for the datacast application ($Datacast \rightarrow S8$).

For this reason, a second exploration with extended search space is performed. More precisely, to relax the negative impact of the bursty instruction fetches from shared memory by the CPU, a traffic shaper is inserted into the system, and its time-out value is added to the search space of design space exploration. This measure leads to the discovery of feasible system configurations. Figure 8 visualizes the system behavior with memory access priorities $inst\_fetch > frame\_buf1 > frame\_buf2$ and increasing time-out values enforced by the inserted traffic shaper.

Clearly, the datacast application suffers from the traffic shaping delay. However, its deadline is only violated for time-out values larger than 55 ms. The frame processing application, on the other hand, profits from the relaxed worst-case instruction fetch burst: its latency decreases with increasing time-out values, and falls below the deadline for time-out values larger than 45 ms. Consequently, feasible (Pareto-optimal) system configurations are obtained for time-out values between 45 and 55 ms.

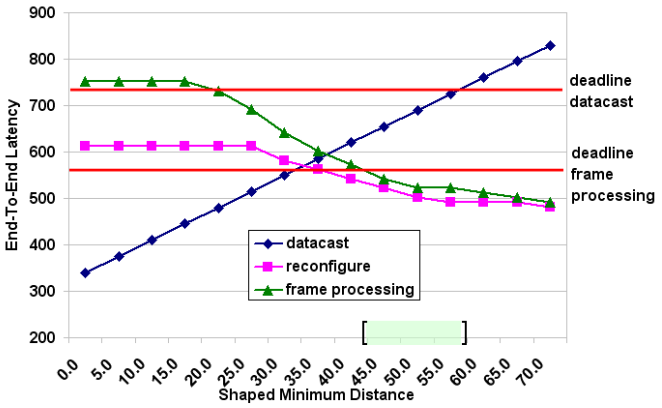

Fig. 8. Impact of Traffic Shaping on System Performance. Shared memory access priorities: $inst\_fetch > frame\_buf1 > frame\_buf2$. The interval of feasible solutions ranges from timeout values 45 to 55.

D. System dimensioning

After determining the shaper's time-out values corresponding to feasible system configurations, we perform, for several Pareto-optimal solutions, the sensitivity analysis of the clock frequencies of the processing elements. Figure 9 shows the speed factors by which the initial resource clock frequencies can be scaled down, without jeopardizing the system performance. Notice that, during the sensitivity analyis only one resource has been modified at a time.

Based on these values, the designer may now choose a feasible solution, that delivers the maximum robustness with respect to system variations or future design modfications.

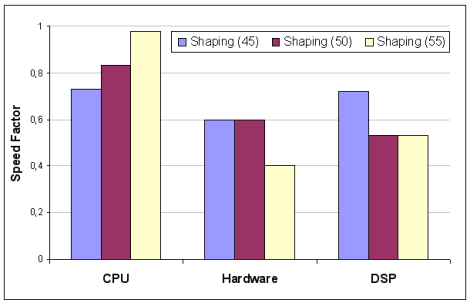

Fig. 9. The minimum speed factors of the hardware components corresponding to three different feasible system configurations

VII. Conclusion

In this paper we have given an overview over state-of-the-art modular performance verification techniques for distributed systems. Furthermore, we have highlighted specific timing implications that require attention when addressing multiprocessor-system-on-chip setups. For this reason, we have presented recent extensions that leverage the applicability of such methods also in the MpSoC domain.

By means of a comprehensible example, we have demonstrated that high level modeling and formal performance analysis are adquate tools for the verification, optimization and dimensioning of heterogneous multiprocessor systems.

References

[1] B. Akesson, K. Goossens, and M. Ringhofer. Predator: a predictable SDRAM memory controller. *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 251–256, 2007.

[2] S. Bleuler, M. Laumanns, L. Thiele, and E. Zitzler. PISA - A Platform and Programming Language Independent Interface for Search Algorithms. In *Proc. of the Conference on Evolutionary Multi-Criterion Optimization (EMO)*, Faro, Protugal, April 2003.

[3] J. L. Boudec and P. Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer, 2001.

[4] S. Chakraborty, S. Künzli, and L. Thiele. A General Framework for Analysing System Properties in Platform-Based Embedded System Designs. In *Proc. of the IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*, Munich, Germany, 2003.

[5] W. Haid and L. Thiele. Complex Task Activation Schemes in System Level Performance Analysis. In *Proc. of the IEEE/ACM International Conference on HW/SW Codesign and System Synthesis (CODES-ISSS)*, Salzburg, Austria, September 2007.

[6] A. Hamann, M. Jersak, K. Richter, and R. Ernst. A Framework for Modular Analysis and Exploration of Heterogeneous Embedded Systems. *Real-Time Systems Journal*, 33(1-3):101–137, July 2006.

[7] A. Hamann, R. Racu, and R. Ernst. A Formal Approach to Robustness Maximization of Complex Heterogeneous Embedded Systems. In *Proc. of the IEEE/ACM International Conference on HW/SW Codesign and System Synthesis (CODES-ISSS)*, Seoul, South Korea, October 2006.

[8] S. Heithecker and R. Ernst. Traffic shaping for an FPGA based SDRAM controller with complex QoS requirements. *Proceedings of the 42nd annual conference on Design automation*, pages 575–578, 2005.

[9] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System Level Performance Analysis - The SymTA/S Approach. *IEE Proceedings Computers and Digital Techniques*, 152(2):148–166, March 2005.

[10] T. Henzinger and S. Matic. An Interface Algebra for Real-Time Components. In *Proc. of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2006.

[11] J. Lehoczky. Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines. In *Proc. of the IEEE Real-Time Systems Symposium (RTSS)*, 1990.

[12] R. Racu, A. Hamann, and R. Ernst. Automotive System Optimization using Sensitivity Analysis. In *International Embedded Systems Symposium (IESS), Embedded System Design: Topics, Techniques and Trends*, pages 57–70, Irvine, CA, USA, June 2007. Springer.

[13] R. Racu, A. Hamann, and R. Ernst. Sensitivity Analysis of Complex Embedded Real-Time Systems. *to appear in Real-Time Systems Journal*, early 2008.

[14] S. Schliecker, M. Ivers, and R. Ernst. Integrated analysis of communicating tasks in MPSoCs. *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, pages 288–293, 2006.

[15] S. Schliecker, M. Ivers, and R. Ernst. Memory Access Patterns for the Analysis of MPSoCs. *Circuits and Systems, 2006 IEEE North-East Workshop on*, pages 249–252, 2006.

[16] J. Staschulat and R. Ernst. Worst case timing analysis of input dependent data cache behavior. *Euromicro Conference on Real-Time Systems*, 2006.

[17] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm for Multiobjective Optimization. In *Proc. Evolutionary Methods for Design, Optimisation, and Control*, pages 95–100, Barcelona, Spain, 2002.